



TITLE:

Standard declarations (算法言語の設計-記述-処理の研究 : ALGOL N)

AUTHOR(S):

IGARASHI, S.; IWAMURA, T.; SAKUMA, K.; SHIMAUTI, T.; SHIMIZU, T.; TAKASU, S.; WADA, E.; YONEDA, N.

CITATION:

IGARASHI, S. ...[et al]. Standard declarations (算法言語の設計-記述-処理の研究 : ALGOL N). 数理解析研究所講究録 1969, 66: 66-74

ISSUE DATE:

1969-02

URL:

<http://hdl.handle.net/2433/107886>

RIGHT:

66

6. Standard declarations

This section describes the ordered set SD of standard declarations supporting the use of the language. The description takes the form of orderly enumeration of a family of disjoint subsets in SD .[†] The description of each member of this family is headed by a reference numbering of the form $(SDi.j.k)$. The ordering in SD is such that one element under $(SDi_1.j_1.k_1)$ precedes another element under $(SDi_2.j_2.k_2)$ if $(i_1.j_1.k_1)$ precedes $(i_2.j_2.k_2)$ in the lexical order.

Each member description begins either with a '{' (not a '{') or with an 'f'.

(a) If a member description begins with a '{', then the matching '}' terminates that description, and the member has a single element, which is the enclosed text.

(b) If a member description begins with an 'f', then the part enclosed by the first following '{' and the matching '}', with which the member description is terminated, gives an element of the member when asterisked symbols (T^* for example) in that part are consistently replaced by certain objects as prescribed after the leading 'f' in terms of common mathematical notations ($T^*e T$ for example).

†

In the present document, instead of the whole family covering SD , only some illustrating subfamily will be presented. Other members, including those for input/output operations, will be found elsewhere.

6.1. Basic operations

6.1.1. Copying and enproceduring operations

(SD1.1.1) {let copy operate before all left after copy right}

(SD1.1.2) {let enproc operate before all left after all right}

(SD1.1.3) $\int T^*eT$ {let copy() represent procedure(T*)T* by
 ((original) code (c:original, d:T*) T* by
 (core(Q) let w(Q)={<c:,Q₁>,<d:,Q₂>};
 m(Q₂)←m(Q₁); w(Q₂)←w(Q₁); ⇒ Q₂ end of core))}

(SD1.1.4) $\int T^*eT$ {let enproc() represent
procedure(T*)procedure()T* by
 ((body) procedure()T* by
 (() body))}

6.1.2. Simple assignment operations

(SD1.2.1) {let ←, := operate before ←, := left after all right}

(SD1.2.2) $\int T^*e\{\text{real, bits, } \overset{\text{string}}{\text{reference}}\}^U T$ procedure, Δ*e{←, :=}
 {let ()Δ*() represent procedure(T*,T*)effect by
 ((destination, source)
 code (d:destination, s:source) effect by
 (core(Q) let w(Q)={<d:,Q₁>,<s:,Q₂>};
 w(Q₁)← p(m(Q₁),w(Q₂)); ⇒ Q₀ end of core))}

6.1.3. Simple comparison operations

(SD1.3.1) {let =, ≠ operate before left after ←, := right}

(SD1.3.2) {let <, ≤, ≥, > operate before =, ≠ left
after ←, := right}

(SD1.3.3) $\int T^*eT, \Delta^*e\{=, \neq\}; T^*=\text{real}, \Delta^*e\{<, \leq, \geq, >\}$
 {let ()Δ*() represent procedure(T*,T*)bits by
 ((left,right) code (l:copy left, r:right, t:l) bits by
 (core(Q) let w(Q)={<l:,Q₁>,<r:,Q₂>,<t:,Q₃>};
 if w(Q₁)Δ*w(Q₂) then ⇒Q₃, else →next;
 w(Q₃)← 0; ⇒Q₃ end of core))}

6.1.4. Conditional operations

(SD1.4.1) {let if operate before all left}(SD1.4.2) {let then operate}(SD1.4.3) {let else, do operate after all right}

(SD1.4.4) $\int T^*eT$ {let if()then()else() represent
procedure(bits, T*, T*) T* by
 ((condition, then, else)
code (c:condition, t:enproc then, e:enproc else)
enproc T* by
 (core(Q) let w(Q)={<c:,Q₁>,<t:,Q₂>,<e:,Q₃>};
 if the bit string w(Q₁) contains at least one 1
 then $\Rightarrow Q_2$, else $\Rightarrow Q_3$ end of core) ())}

(SD1.4.5) {let if()do() represent procedure(bits, effect)effect
by((condition, statement) if condition then statement
else dummy)}

6.1.5. Basic arithmetic operations

(SD1.5.1) {let +,~ operate before =,≠,<,≤,≥,> left
after +, := right}

(SD1.5.2) {let ×,/ operate before =,≠,<,≤,≥,>,+,- left
after +, := right}

(SD1.5.3) $\int \Delta^*e\{+,-,\times,/ \}$ {let () Δ^* () represent
procedure(real,real)real by ((left,right)
code (a:copy left, b:right) real by
 (core(Q) let w(Q)={<a:,Q₁>,<b:,Q₂>};
 if the arithmetic operation meant by w(Q₁) Δ^* w(Q₂)
 can be performed then \rightarrow next, else $\Rightarrow L0$;
 let W₃ be the resulting value of that operation
 (possibly with some implementation dependent deviation);
 w(Q₁) \leftarrow p(m(Q₁),W₃); $\Rightarrow Q_1$ end of core)}

(SD1.5.4) {let -() represent procedure(real)real by
 ((right) begin (right)b; (copy b)a; a:=0; a-b end)}

(SD1.5.5) {let /() represent procedure(real)real by
 ((right) begin (right)b; (copy b)a; a:=1; a/b end)}

6.1.6. Some enquiry operations

- (SD1.6.1) {let mode, length, bd operate before all left
after right}
- (SD1.6.2) {let mode() represent procedure(real)structure
(fix:bits, min:real, step:real, max:real) by ((real)
begin (real)x; code (fix:1, min:copy x, step:real,
max:copy x) (mode real) by
(core(Q) let w(Q)={<fix:Q₁>, <min:Q₂>, <step:Q₃>, <max:Q₄>};
let M stand for m(Q₂);
w(Q₂)← the minimum value in W_M;
w(Q₄)← the maximum value in W_M;
if M=real [precision R] with a real number R,
then →next, else →Kfixed;
w(Q₃)← p(m(Q₃), R); w(Q₁)← 0; ⇒Q;
Kfixed: let M=real [R₁:R₂:R₃] with real numbers R₁,
R₂, R₃;
w(Q₃)← p(m(Q₃), R₂); ⇒Q end of core) end)}
- (SD1.6.3) {T*e{bits,string} let mode() represent
procedure(T*)structure (exact:bits, length:real) by ((some)
code (s:some, t:(exact:1, length:)) (mode T*) by
(core(Q) let w(Q)={<s:Q₁>, <t:Q₂>}; integer
let w(Q₂)={<exact:Q₃>, <length:Q₄>};
let m(Q₁)=T* [Y I] ,where I is an integer and Y
is either exact or varying ;
w(Q₄)← p(m(Q₄), I);
if Y=exact then ⇒Q₂, else →next;
w(Q₃)← 0; ⇒Q₂ end of core) }
- (SD1.6.4) {T*eT let mode() represent procedure(array T*)
structure (lbd:real, ubd:real) by ((array)
code (a:array, t:(lbd:1, ubd:)) (mode array T*) by
(core(Q) let w(Q)={<a:Q₁>, <t:Q₂>}; integer
let w(Q₂)={<lbd:Q₃>, <ubd:Q₄>};
let m(Q₁)=array [I₁:I₂] T* with integers I₁, I₂;
w(Q₃)← p(m(Q₃), I₁); w(Q₄)← p(m(Q₄), I₂);
⇒Q₂ end of core) }

6.1.6 continued

(SD1.6.5) $\int T^*e\{\text{bits}, \text{string}\}$ {let length() represent
procedure(T*)real by ((string) code (s:string,t:) real by
 (core(Q) let w(Q)={<s:,Q₁>,<t:,Q₂>}; integer
 let I be the integral length of w(Q₁);
 w(Q₂)← p(m(Q₂),I); \Rightarrow Q₂ end of core)}

(SD1.6.6) $\int T^*eT$ {let bd() represent procedure(array T*)real
by ((array) (mode array)[lbd:])}

(SD1.6.7) $\int T^*eT$ {let ()bd represent procedure(array T*)real
by ((array) (mode array)[ubd:])}

6.2. Extended operations

6.2.1. Repetitive operations

- (SD2.1.1) {let while,until operate before left after all right}
- (SD2.1.2) {let succ,step operate before left
after +, :=, =, ≠, <, ≤, ≥, >, +, -, ×, / right}
- (SD2.1.3) $\int T^* \epsilon T$ {let ()succ() represent procedure(T^*, T^*)
structure (init:enproc T^* , succ:enproc T^*) by
((a,s) (init:enproc a , succ:enproc s)) }
- (SD2.1.4) $\int T^* \epsilon T$ {let ()while() represent procedure($T^* \text{succ} T^*, \text{bits}$)
structure (init:enproc T^* , succ:enproc T^* , while:enproc bits) by
((r,t) begin (r rl; (init:copy rl[init:],
succ:copy rl[succ:], while:enproc t) end) }
- (SD2.1.5) {let ()step() represent procedure(real,real)
structure (init:enproc real, step:enproc real) by
((a,b) (init:enproc a , step:enproc b)) }
- (SD2.1.6) {let step() represent procedure(real)(real step real)
by ((b) 1 step b) }
- (SD2.1.7) {let ()until() represent procedure(real step real,real)
structure (init:enproc real, step:enproc real,
until:enproc real) by
((progression,limit)
begin (progression)ab; (init:copy ab[init:],
step:copy ab[step:], until:enproc limit) end) }
- (SD2.1.8) {let ()until() represent procedure(real,real)
(step real until real) by ((a,c) a step 1 until c) }
- (SD2.1.9) {let until() represent procedure (real)(real until real)
by ((c) 1 until c) }

6.2.1 continued

(SD2.1.10) {let from operate before all left}

(SD2.1.11) $\int T^*eT$ {let ()from()^()do represent
procedure(T*, T*succT*while bits, effect) effect by
 ((cvar, domain, statement)
begin (cvar)cv; (domain)dom; cv← dom[init:]();
 next: if dom[while:]() do
begin statement; cv← dom[succ:](); go to next end
end)}

(SD2.1.12) {let ()from()do() represent
procedure(real, until real, effect) effect by
 ((cvar, domain, statement)
begin (cvar)cv; (copy cv)v1; (copy domain)dom;
 (copy dom[init:]())a1; (copy dom[step:]())b1;
 (copy dom[until:]())c1;
 v1 from a1 succ v1+b1 while if b1>0 then v1≤c1
else if b1<0 then v1≥c1 else v1≠c1
do begin cv:=v1; statement end
end)}

6.2.2. Assign-and-hold operations

(SD2.2.1) {let the operate before all left after right}

(SD2.2.2) $\int T^*eT$ {let the() represent
procedure(T*) procedure(effect) T^* by
 ((expression) procedure(effect) T^* by ((dummy) expression))}

(SD2.2.3) $\int T^*eT, \Delta^*e\{\leftarrow, :=\}$ {let () $\Delta^*()$ represent
procedure(the $T^*, T^*)$ T^* by ((the destination, source)
begin (the destination(dummy))destination;
 destination Δ^* source; destination
end)}

6.2.3. Reference handling operations

(SD2.3.1) {let ref, has type, as type operate before all left
after right }

(SD2.3.2) $\int T^* \in T$ {let ref() represent procedure(T*) reference
by ((referent) code (q:referent, r:reference) reference
by (core(Q) let w(Q)={<q:,Q₁>,<r:,Q₂>} ;
w(Q₂)← {Q₁}; \Rightarrow Q₂ end of core))}

(SD2.3.3) $\int T^* \in T$ {let () has type() represent
procedure(reference, T*) bits by ((ref,type)
code (r:ref, t:type, b:0) bits by
(core(Q) let w(Q)={<r:,Q₁>,<t:,Q₂>,<b:,Q₃>};
if w(Q₁)= empty, then \Rightarrow Q₃, else \rightarrow next;
let w(Q₁)={Q₄};
if t(Q₂)=t(Q₄) then \rightarrow next, else \Rightarrow Q₃;
w(Q₃)← 1; \Rightarrow Q₃ end of core))}

(SD2.3.4) $\int T^* \in T, T^* \neq \text{reference}$ {let () as type() represent
procedure(reference, T*) T* by ((ref, type)
code (r:ref, t:type) T* by
(core(Q) let w(Q)={<r:,Q₁>,<t:,Q₂>};
if w(Q₁)=empty, then \Rightarrow Q₂, else \rightarrow next;
let w(Q₁)={Q₄};
if t(Q₂)=t(Q₄) then \Rightarrow Q₄, else \Rightarrow Q₂ end of core))}